Porting a Vector Library: a Comparison of MPI, Paris, CMMD and PVM (or, "I'll never have to port CVL again")

Jonathan C. Hardwick

November 1994

CMU-CS-94-200

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

An earlier version of this paper appeared in "Proceedings of the 2nd Scalable Parallel Libraries Conference", Mississippi State University, Mississippi, October 1994.

Abstract

This paper describes the design and implementation in MPI of the parallel vector library CVL, which is used as the basis for implementing nested data-parallel languages such as NESL and Proteus. We compare the ease of writing and debugging the portable MPI implementation of CVL with our experiences writing previous versions in CM-2 Paris, CM-5 CMMD, and PVM, and give initial performance results for MPI CVL running on an IBM SP-1, Intel Paragon, and TMC CM-5.

This research was sponsored in part by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. It was also supported in part by the Pittsburgh Supercomputing Center (Grant ASC890018P), who provided TMC CM-2 time, and in part by the National Center for Supercomputing Applications (Grant TRA930102N), who provided TMC CM-5 time, and in part by the Argonne National Laboratory, who provided IBM SP-1 time. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Wright Laboratory or the United States Government.

Keywords: CVL, MPI, NESL, parallel language, vector library, nested data parallelism, segmented operations

1 CVL overview

CVL (C Vector Library [5]) is a library of vector functions callable from C. It provides an abstract vector memory model that is independent of the underlying architecture, and is designed so that efficient CVL implementations can be developed for a wide variety of parallel machines. Machine-specific versions currently exist for the Thinking Machines CM-2 and CM-5, the Cray Y-MP and Y-MP/C90, and the MasPar MP-1 and MP-2 [11], as well as a portable serial version written in ANSI C.

CVL was developed primarily as the lowest-level component of a three-tier system that implements the portable nested data-parallel language NESL [3], although it is also used as a stand-alone C vector library, and as a back end by the Proteus language [15]. Figure 1 shows an overview of the current NESL system. The shaded components (CVL and its parallel implementations) are the main focus of this paper.

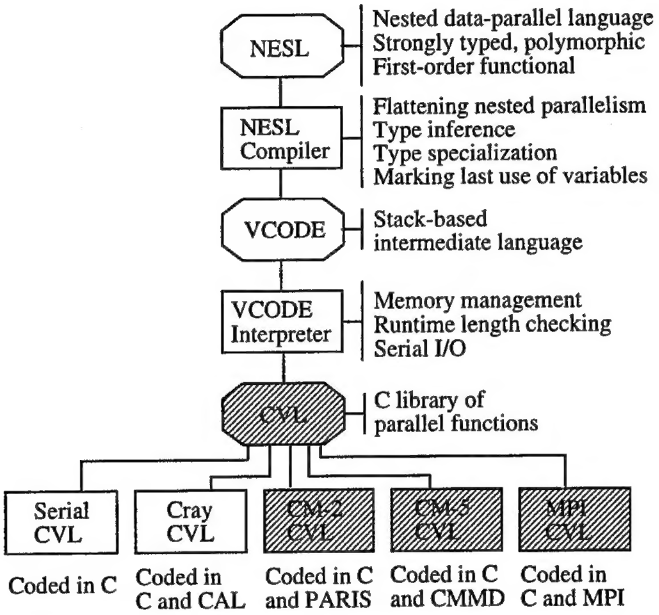

Figure 1: Overview of the NESL system

NESL is a parallel functional language with an ML-like syntax, and is designed to be used interactively. It is the first language to support fully nested data-parallelism: that is, parallelism both within nested data structures and across nested data-parallel function calls. This enables it to combine the advantages of data-parallel and control-parallel languages. NESL's basic data type is a vector, and nested parallelism is achieved through the ability to apply functions in parallel to each element of a vector. For example, a sparse array can be represented as a vector of rows, with each row represented by a subvector that contains the nonzero elements (and corresponding indices) in that row. A parallel function that sums the elements of a vector can then be applied in parallel to sum each row of this sparse array. This ability to operate efficiently on irregular data structures is one of NESL's main strengths [6].

NESL is compiled into VCODE [4], a stack-based intermediate language. NESL's nested data structures are "flattened" into segmented vectors [2], allowing a single function call to operate on the entire data structure at once. The resulting VCODE is then interpreted, with the interpreter using CVL to achieve portability and efficiency. Since each VCODE instruction typically translates to a single CVL function call operating on vectors, the overhead of interpreting each instruction is amortized over the length of its vector operands [6].

To support high-level languages, CVL supplies a rich variety of vector functions, including elementwise function application, global communication functions such as scans and permutations, and ancillary functions such as timing and memory allocation. This specialization of functionality allows CVL to be tuned for each machine to which it is ported. For example, the segmented scan and reduction operations on the Cray have been vectorized by hand in Cray assembler language using a novel algorithm [9], and are much faster than could be easily achieved using Fortran or C. Most of the vector functions are supplied in both unsegmented and segmented versions, with the segmented versions being used to implement nested data parallelism, while the unsegmented functions typically run faster and can be used in the special case of a single-segment vector.

The CVL functions whose implementations are likely to vary most between different machine architectures can be broken down into two categories:

Scans and Reductions These functions apply an associative combining operator such as addition or maximum across a vector, returning either a single value (reduction), or a vector containing the "running totals" (scan, or parallel prefix). Their implementation on a parallel machine is normally via a combining tree, either in hardware (CM-2 and CM-5) or in software (MPI).

Permutations CVL has an extensive set of functions which permute the elements of a vector into a new vector. They are specialized by type, segmentation, direction, mapping, and default behavior. Their implementation is complicated by the fact that the resulting communication pattern is data-dependent and therefore may not be balanced.

Note that because of CVL's structure as a library of specialized single-purpose functions, we cannot take advantage of some of the data locality optimizations that would be possible if we were to use a compiled language as the back end of the system. For example, a vector multiply-add operation is implemented in CVL as a vector add followed by a vector multiply, resulting in two loops over the data instead of the optimal one. All CVL functions have this "single loop" style, and the effects of the resulting poor data cache locality can be seen in Section 4. During CVL's design, this loss of efficiency was accepted as the price to pay for a simple, portable, parallel library. At the cost of extra complexity, more specialized functions such as a vector multiply-add operation could be added to CVL. It is worth noting that for the irregular problems for which NESL is designed, programs written in conventional languages such as Fortran also have poor data locality.

The rest of the paper is organized as follows. In Section 2, we outline the decisions to be made when porting CVL to a new platform. In Section 3 we describe the CM-2, CM-5, PVM, and MPI implementations in terms of how these decisions were made for the different platforms, and discuss their ease of development and relative performance. In Section 4 we offer some comparative performance results for low-level CVL primitives. Finally in Section 5, we offer some conclusions and recommendations for future work based on these results.

2 Porting CVL: the choices

There are six main choices to make when porting CVL to a new platform:

Language and Library The choice of language (and communication library, if one is used) for a CVL implementation obviously affects its ease of development and final performance. This decision has an impact on most of the other decisions listed below. All existing parallel implementations of CVL are written in C with calls to a communication library. If we use a language that does not follow the C calling conventions, we'll need extra "wrapper" functions.

Data Distribution All CVL data is stored in a global area of "vector memory", which can only be accessed and modified through CVL routines. We must choose a method of distributing this memory across the machine. All existing parallel implementations of CVL use a balanced block distribution, in which every vector is distributed equally across the nodes.

Vector Representation To enforce the abstraction of vector memory, CVL functions use parameters with the C type `vec_p` as abstract handles to vectors. We must choose an internal representation of this `vec_p` type, with the restriction that CVL vectors may be reused to store data of different types. Thus, if a `vec_p` is represented by a pointer, the area of memory pointed to must be maximally aligned on the target architecture.

Segment Representation CVL defines a segmented vector as an unsegmented vector that contains the actual data, plus a "segment descriptor" that describes how to partition the unsegmented vector into subvectors. The separation of data from its structure enables elementwise functions to be oblivious to the structure of their operands, and allows sharing of both data vectors (between segmented vectors that have the same data but different segmentation) and segment descriptors (between segmented vectors that have the same segmentation but different data).

Segment descriptors are stored in vector memory, and have two lengths associated with them: the number of segments and the total number of elements in the vector. They must also describe which elements are in which segments; the representation of this description is the main implementation decision. A simple representation, such as a vector of segment lengths, saves space but can involve additional work when performing segmented functions. A more complicated representation might add a vector containing the offset of the beginning of each segment. This is redundant information, so precomputing it and adding it to the segment descriptor is a space/time tradeoff. This tradeoff can result in better performance if the segment descriptor will be used two or more times by CVL functions that would otherwise have to compute that information.

Host Model Depending on the machine architecture, CVL may be implemented in either a hostless or host/node style, corresponding roughly to the SPMD and SIMD models of parallel computation.

In the hostless style, a complete copy of CVL (and the associated user program) runs on each node. This has the advantage of minimizing synchronization overhead—nodes can run independently on sequences of CVL instructions which involve no communication. However, since CVL has a single thread of control and serial input/output semantics, one node is designated to perform all input/output operations.

In the host/node style, only the host machine runs the user program (note that the host may be a front-end computer attached to a parallel machine, or a chosen node of the parallel machine). CVL on the host is reduced to a library of function stubs that broadcast function calls and arguments to CVL slave processes running on the nodes. This has several advantages. First, the slave process on each node is small, leaving more room for user data. Second, the host and nodes can overlap computation; the user program on the host can continue execution while the nodes execute CVL instructions. Finally, this may be the only way to give the user program a single-process, Unix-style environment.

Message Buffering CVL vector permutation functions can be naively implemented as very fine-grained all-to-all communication between the nodes of a parallel machine. If CVL is implemented on top of a conventional message-passing system, aggregating vector elements into a buffer before sending will almost certainly improve overall performance, since message overhead is amortized over multiple elements being sent between the same two nodes.

There may be an additional performance benefit to be gained by aggregrating elements in fixed-size buffers in user space, rather than relying on system buffering [17]. However, this introduces the problem of how nodes determine when all messages have been sent and received. It is not enough for the nodes to use a barrier to agree that all messages have been sent, since messages may still be in transit. There are at least two well-known solutions. The first is to require each node to send at least one message to every other node, with the final message between two nodes having a termination flag set. The second solution is for all nodes, once they have sent all their messages, to intersperse attempts to receive messages with a global summation of the number of messages sent by each node minus the number it has received, terminating when the total falls to zero.

3 Existing CVL implementations

In this section we describe the machine-specific implementations of CVL for the CM-2 and CM-5, the prototype PVM implementation, and the portable MPI implementation. For each implementation, we discuss the particular solutions chosen to the CVL porting decisions outlined in Section 2, and their impact on development time and final performance.

3.1 CM-2 CVL

The implementation of CVL for the TMC CM-2 is written in C and Paris [18], a parallel instruction set for the CM-2's SIMD processing array. CM Fortran [20] was not used because at the time it did not have the ability to alias subsections of arrays (for example, storing a vector of integers into the middle of where a vector of floating-point numbers used to be), and therefore did not meet CVL's vector reuse requirements.

CVL vector memory maps naturally to the Paris concept of fields, and a **vec_p** is implemented as a Paris field ID. Paris has direct support for segmented functions, so the segment descriptor is designed around the segmentation format that Paris expects. Specifically, a CM-2 CVL segment descriptor is implemented as three vectors: the first contains the number of elements in each segment, the second has a bit set for every segment that is empty (used to avoid fencepost errors in certain segmented functions), and the third has a bit set for every element that is at the beginning of a segment (used by segmented functions that need to reset their behavior at the beginning of each segment).

The CM-2 is a SIMD machine with an explicit host/node model, so the host model decision is made for us. Finally, Paris supplies its own permutation primitives, rather than explicit message-passing functions, so no buffering is needed.

Paris ease of development: Paris has direct equivalents for many of the CVL functions, simplifying their implementation. However, development and debugging was hampered by three common problems of using a remote shared supercomputer:

- Unfamiliar and limited debugging tools.

- Competition for editing and compilation resources with other users on the front end machine, and for machine resources on the supercomputer.
- A limited account allocation that discourages the use of run-time debugging.

Debugging therefore tended to be of the "core dump and printf" style.

Paris performance: There are several performance penalties involved in using Paris:

- Paris uses the older "fieldwise" representation of data, requiring extra transpose operations to use the CM-2's floating-point accelerators. The newer and more efficient "slicewise" representation that is now available in CM Fortran [16] is not supported by Paris.
- Paris communication functions use one virtual processor per element, which requires that the user make "aliases" of dynamically-sized fields before passing them to a communication function. However, aliases can only be made of originally allocated fields, and not of fields constructed by pointer manipulation. Since CVL does not know whether a particular `vec_p` is original or constructed, it must copy each vector into a freshly-allocated temporary field before making an alias of the temporary and using it in a communication function.
- Paris fields are restricted to 64 kbits per processor in length, whereas a CM-2 typically has 256 kbits or 1024 kbits installed. This imposes an artificial limit on the maximum size of problems that can be solved with CM-2 CVL.

CM-2 CVL is therefore not fully exploiting the machine in terms of floating point performance, memory traffic, or memory utilization. In a comparison on the CM-2 between CM Fortran and NESL, CM Fortran wins on benchmarks that emphasize floating-point operations, whilst NESL wins on benchmarks that utilize nested parallelism and emphasize communication [6].

3.2 CM-5 CVL

The implementation of CVL for the TMC CM-5 is written in C and the message-passing library CMMD [21]. Again, CM Fortran could not be used because at the time it lacked array aliasing capabilities. To avoid possible confusion with the portable MPI CVL implementation running on a CM-5, we shall henceforth refer to CM-5 CVL as CMMD CVL.

CMMD CVL uses a balanced block data distribution, with `vec_p`'s implemented as 64-bit-aligned pointers. Since the nodes involved in a single CM-5 program run identical code (and have no virtual memory system), a given `vec_p` can have the same value across all nodes. CMMD CVL segment descriptors contain two vectors and two scalars. The first vector contains the index of the start of each segment, with the high bit indicating whether the segment is empty. The second contains the number of segments that start at each element; note that this is cannot be represented by a simple bit, since some of the segments may be empty (allowing for the possibility of empty segments is probably the single greatest cause of complexity in CVL implementations). The two scalars are the number of the segment containing the first element stored on the node, and the index of the start of that segment. These are used in segmented permutation functions, where the permutation happens within a segment, and a local offset within a segment must be converted into a global offset within the vector as a whole.

The CM-5 may be programmed in either a hostless or host/node style, and the initial CVL implementation used the hostless style to speed development. However, the semantics of VCODE were later extended to include the ability to spawn subprograms (such as an X11 interface). Since the CM-5 does not run a full Unix-style operating system on each node, there was no way to support this ability using the hostless model. CMMD CVL was therefore rewritten to use the host/node model. The CM-5's dedicated control network means that the overhead of broadcasting each instruction in the host/node model is minimal compared to the cost of executing it. Benchmarks at the time showed that the host/node implementation was in fact marginally faster than the hostless implementation for large NESL problems, probably due to overlap in computation between the VCODE interpreter on the host and the CVL slave processes on the nodes.

CMMD CVL permutation functions are written using the CMAM active message subset of CMMD. With one word reserved for the active message function pointer, this restricts the message payload to four 32-bit words. Therefore, for most permutations, no buffering is used: vector elements are sent to the new destination in a single message as soon as the correct node and offset have been calculated. The sole exception to this is the integer "pack" function, where an offset and up to three consecutive integers to be placed at that offset are sent in one packet. Pack is called frequently by the VCODE interpreter to load-balance vectors across the machine, so this is a worthwhile optimization. Note that we cannot repeat it for the floating-point pack function, since we cannot fit an offset and more than one 64-bit float into the four available 32-bit words in an active message packet.

The critical inner loop for all message-sending functions contains a machine-specific optimization. Conceptually, a division is used to calculate which node a particular vector offset corresponds to. Rather than dividing by the amount of data on a node, it is faster to precompute the reciprocal of this number, and replace the integer division in the inner loop with a floating-point multiplication by the reciprocal, followed by rounding to an integer.

The global-sum-until-zero method is used to determine when all messages have been received, since the CM-5's control network supports fast global sums. Polling is used to detect the arrival of active messages (rather than interrupts, which have a much higher overhead), so the global sums must be interspersed with calls to a polling routine to receive any messages that have arrived.

CMMD ease of development: CMMD went through three major versions during the writing of CMMD CVL, necessitating constant rewriting and considerably lengthening the development process. Active messages [22] were introduced as a third-party extension to the first major version of CMMD, were incompatible with the second version, and were finally adopted by Thinking Machines in the third version.

CMMD supplies scan and reduction functions in both unsegmented and segmented versions, simplifying the task of implementing CVL. The one CVL combining primitive that CMMD does not provide is multiplication. CMMD CVL therefore implements multiplicative scans and reductions (products) using a standard binary-tree message-passing algorithm. Since this uses the data network rather than the control network, the overhead for a multiply-scan function on 64-bit floating point numbers is approximately 20 times greater than that of the overhead for an add-scan, depending on the number of nodes participating (see also Table 3).

With limited account resources available on the CM-5 being used for development, debugging was again of the "core dump and printf" style, with very little interactive use of the debugger on running programs. However, the ability to recover vector data from node core dumps was a considerable improvement over the CM-2's opaque memory model.

CMMD performance: The choice of C and CMMD on the CM-5 involves computational performance penalties similar to those encountered with C and Paris on the CM-2. When the CM-5's vector units were introduced, offering memory-to-memory floating-point performance an order of magnitude better than that achievable with the node SPARC processors, only CM Fortran was upgraded to support them. Using the vector units from C is possible via a set of assembler macros [19], but with the limited resources available, updating CMMD CVL to use the vector units has not been attempted. Note that a new CVL implementation using the current aliasing capabilities of CM Fortran would be able to fully realize the floating-point performance of both the CM-2 and the CM-5.

3.3 PVM CVL

To explore the possibilities of a more portable CVL, a proof-of-concept implementation [14] of CVL was written in C and PVM 3.0 [13]. A subset of the CVL functions were implemented in order to get an idea of the relative speed and usability of PVM for our application.

To speed development, the design of PVM CVL was mostly copied from that of CM-5 CVL. Since the nodes in a PVM computation are not guaranteed to be identical, and are probably running in a virtual memory environment, a `vec_p` is represented as a 64-bit-aligned offset to be added to a per-process memory base. The host/node style of CM-5 CVL was also adopted, although this was probably a mistake. The intent was that a user would run the host process on their own workstation, with the node processes running on a cluster or supercomputer. However, since PVM's multicast function is actually implemented as a linear series of point-to-point sends, the overhead to broadcast each new instruction was considerable.

PVM CVL's permutation functions relied entirely on system buffering, with each node sending exactly one PVM buffer to every other node. This requires at least as much system buffer space as the sum of the sizes of the messages being sent. An additional problem is that the interconnection network is not used efficiently: there is no network traffic for most of a permutation function, as the individual nodes pack messages into buffers, followed by a sudden burst of traffic as every node tries to send buffers to every other node.

PVM ease of development: In terms of usability, PVM is a great improvement over both Paris and CMMD. Since it runs on workstations, development can be done on the researcher's own machine, eliminating the problems of competition for resources of a shared supercomputer. Furthermore, all the familiar debugging tools are available to the researcher, and can be used to step through a program in real time without worrying about account resources.

Unfortunately, PVM is an evolving research project rather than an agreed-upon standard. Thus, manufacturers' extensions to PVM that improve its performance (such as the Cray T3D's `pvm_fastsend()` function [10]) are not supported by other PVM implementations. A portable PVM version of CVL would therefore either have to sacrifice performance on such platforms, or would require constant updating as new machine-specific PVM implementations appear.

PVM performance: There were sufficient performance problems with PVM CVL to cause its development to be abandoned, although a rewrite using PVM 3.3 or later could solve some of them.

- Using the PVM method of packing elements into a system buffer before sending does not work well for CVL permutations, since we do not know in advance if successive elements in a vector are being sent to the same node. PVM packs are therefore made on single elements, requiring an extra function call for every element transferred. A possible solution to this would be to implement additional user buffering and use PVM 3.3's `pvm_psend()` function to pack and send a user buffer in a single operation.

- Since PVM 3.0 provided no collective functions, scans and reductions were written using a binary-tree message-passing algorithm, where the number of message delays is logarithmic in the number of node. The heavyweight PVM messages mean that these scan and reduction functions carry a large overhead. Note that the reduction functions provided in PVM 3.3 are not a scalable solution to this problem, since PVM's support of dynamic groups forces it to use a single group server to decide which processes are in a group at any one time. This imposes a serial bottleneck on PVM's collective functions.

- Even in PVM 3.3, support for asynchronous communication to overlap computation and communication and to reduce system buffering remains incomplete [17]. This results in needless synchronization and memory traffic.

3.4 MPI CVL: portability and performance?

Development of MPI CVL was started in the hopes that MPI [12] would combine the advantages of PVM (portability, ease of development) with performance approaching that of existing vendor communication libraries for MPPs. It also promised a more stable specification than that provided by CMMD or PVM.

The design of MPI CVL incorporates lessons learned from the CMMD and PVM CVL implementations. In particular, data distribution and pointer representation are identical to those of PVM CVL, the segment descriptor format is similar to that of CMMD CVL and most of the function implementations were based on those for CMMD CVL.

Since broadcasting will probably be relatively expensive on many of the message-passing machines on which MPI CVL will run, the hostless model was chosen. This limits the capabilities of programs using MPI CVL on machines without a Unix-style operating system on each node, and in particular means that the VCODE interpreter cannot spawn subprocesses on such machines. A host/node implementation for machines with a fast broadcast (for example, the CM-5 and shared-memory multiprocessors) is a possibility for future work.

The segment descriptor layout for MPI CVL is the most complex so far, consisting of the length and starting index of each segment, each element's segment number, and the number and starting index of the segment holding the first element on a node.

Permutation functions are implemented using MPI's nonblocking asynchronous sends and receives. Since MPI doesn't support fine-grain communication, message overhead is amortized over many individual elements by aggregating them in user space buffers. The asynchronous sends and receives enable communication and computation to be overlapped on machines that support the offloading of communication responsibilities from the main processor to a message coprocessor or DMA engine. Every node can therefore have up to four user-space message buffers for every other node; one being packed and one being unpacked by the main processor, and one being sent and one being received by the message coprocessor (see Figure 2).

As in CMMD CVL, a global sum is used to decide when all messages for a given permutation have been sent and received.

MPI ease of development: MPI inherits from PVM all the ease-of-use advantages stemming from the ability to develop code on a local machine. All MPI CVL development was done on a workstation, with final ports to an IBM SP-1, Intel Paragon and TMC CM-5 taking less than a day each (the bulk of this time was spent iteratively refining the code to pass the idiosyncrasies of each machine's C compiler). This compares with the months of effort taken to write the machine-specific CVL implementations for the CM-2 and CM-5.

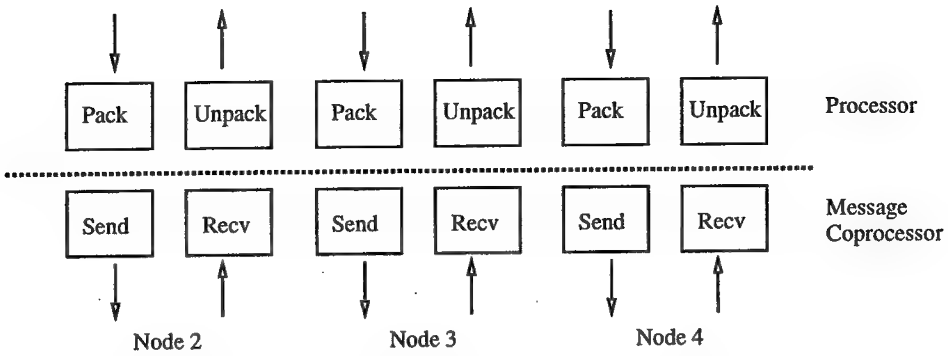

Figure 2: Node 1's buffers on a 4-node machine with message coprocessors

In terms of features, MPI's support for nonblocking asynchronous sends and receives is very welcome, as is the provision of scans and reductions and their extensibility with user-defined combining functions. However, the definition of MPI's scans as inclusive rather than exclusive is annoying, necessitating extra communication to generate exclusive scans for operators with no inverse (for example, a maximum-scan).

MPI performance: Some performance figures for MPI CVL are given in Section 4. Currently, tuning MPI CVL for a particular MPI/machine combination is limited to choosing a size for the message buffers, choosing how often to poll for incoming messages (strongly related to buffer size), and deciding whether or not to use asynchronous sends and receives. Larger message buffers typically allow greater bandwidth, but consume memory that could otherwise be used for user data. The right tradeoff depends on the amount of memory on individual nodes, the number of nodes in the machine, and the shape of the buffer-size/bandwidth curve. The shape of this curve for an SP-1, Paragon, TMC CM-5 for a sample CVL permutation function is shown in Figure 3.

Since the buffer space required per node is proportional to the number of nodes, this scheme is not scalable to thousands of nodes. However, it is reasonable given the machine sizes and node memories in normal use today (how reasonable it will be in the future depends on the number of nodes, memory per node, and message overhead of future machines). For example, when using 8 kbyte buffers on 64 nodes of a Paragon, the buffers account for 2 Mbytes on each node, out of a total memory of 16-64 Mbytes per node. Furthermore, it is more scalable than relying on system buffering, which in the pathological case could require half of the memory on each node.

4 Results

In this section we compare the scalability and efficiency of the machine-specific CM-5 and CM-2 CVL implementations with the portable MPI CVL implementation running on an SP-1, Paragon, and CM-5. The ANL/MSU model MPI implementation `mpich` [7] (version 1.0.5) was used for the MPI benchmarks. This is implemented on top of an abstract device interface, easing the task of porting it to new machines, but adding some overhead since it is layered on top of the manufacturer's own message passing system. Note that no effort was made to tune `mpich` for a particular platform using its various compile-time and run-time parameters. Thus, these results should be considered

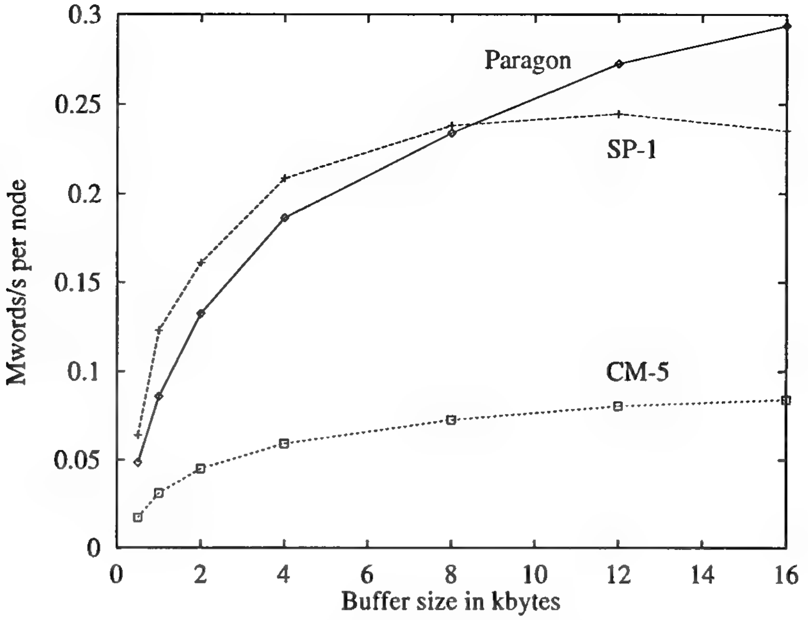

Figure 3: Effect of buffer size on per-node performance of MPI CVL permutation function (32 nodes, unsegmented random permutation of 64-bit words).

preliminary, and performance will probably improve in future. In particular, the SP-1 results were obtained on a modified machine with an experimental version of the vendor message passing library. Details of the machines used are given in Table 1. For the sake of comparison, all benchmarks were made using 8 kbyte message buffers, and do not use asynchronous sends. Except for Figure 5, all benchmarks were run on problem sizes large enough to represent asymptotic memory-to-memory performance, rather than cache-to-cache performance.

Machine	OS	Compiler
IBM SP-1	AIX 3.2.4	mpcc -O2
Intel Paragon	OSF/1 v1.2	icc R4.5 -O3
TMC CM-5	CMOST 7.3	gcc 2.5.8 -O2

Table 1: Machines used for CVL tests

4.1 Scalability of MPI CVL

First, we consider the question of how well MPI CVL scales as the number of nodes is increased. Scalability of elementwise CVL functions with no communication is clearly perfect (modulo startup costs), given MPI CVL's balanced block data distribution. For collective communication operations, the per-element cost is perfectly scalable, and the fixed overhead should scale with the logarithm of the number of nodes, assuming that the particular MPI implementation uses a combining tree algorithm. The main remaining group of CVL functions that might not scale well are the permutations, which involve all-to-all communication. A representative example to show the scalability of an MPI CVL permutation function is shown in Figure 4. This shows the asymptotic per-node performance of a random permutation on an unsegmented vector of 64-bit words, as the number of nodes is varied. The performance of CMMD CVL is also shown.

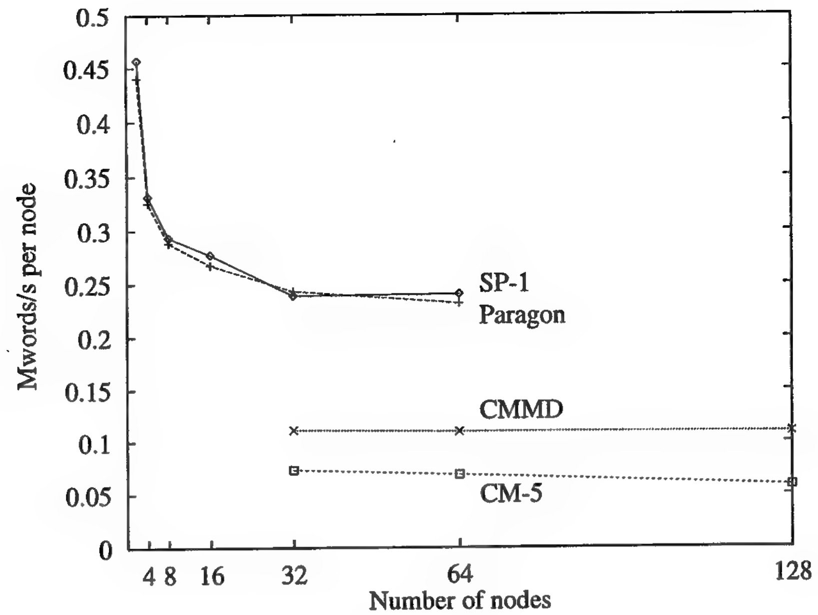

Figure 4: Effect of scaling on per-node performance of CVL permutation function (unsegmented random permutation of 64-bit words). See text for explanation of scaling behavior.

Note that this a graph of permutation performance, rather than network bandwidth: with a random permutation amongst N nodes, $1/N$ of the data remains on a node, artificially raising the apparent performance as the number of nodes decreases. Also, this graph should not be taken as an indication of the scalability of underlying hardware or vendor message systems, since we are not approaching the network bandwidth limits of any of the machines shown (see Section 4.2).

From this graph, the scalability of MPI CVL's permutation functions appears to be quite good on all machines, although on the CM-5 it is not as good as that of CMMD CVL. The permutation performance of MPI CVL on the CM-5 is also significantly less than that of CMMD CVL. This is partly due to the extensive memory traffic necessary to send each element in MPI CVL. For example, in order to prepare one element of an indexed permutation for sending, MPI CVL must load the index value, calculate the node to which it corresponds, load, increment and store the appropriate buffer counter, load the pointer to the appropriate buffer, store the calculated offset in the buffer, and finally load the element value and store it into the same buffer. This is a total of four loads and three stores, plus the amortized cost of the final message send. By comparison, the active message implementation in CMMD CVL has to load the index value, calculate the node to which it corresponds, load the element value, and then perform a function call to the active message send function, for a total of two memory reads and a function call. The extra memory overhead involved in maintaining the buffers in MPI CVL appears to be an unavoidable cost of trying to layer fine-grain communication on top of a coarse-grain message passing system such as MPI.

Table 2 compares the per-node performance of vector permutations from Figure 4 with the per-node performance of 64-bit floating-point vector addition, and adds results for CM-2 CVL. For the CM-2, performance is per "hypernode" (floating-point unit plus 32 associated single-bit processors).

There are several interesting aspects to this table. First, the achievable memory-to-memory computation performance of each of the three MIMD machines is at least an order of magni-

Implementation	MFLOPS	Mwords/s	Ratio
SP-1 MPI	7.9	0.24	33
Paragon MPI	2.6	0.24	11
CM-5 MPI	1.0	0.07	14
CM-5 CMMD	1.0	0.11	9
CM-2 Paris	0.34	0.015	22

Table 2: Per-node performance of CVL 64-bit floating-point vector addition and permutation functions (32 node systems, unsegmented random permutation of 64-bit words).

tude less than its advertised peak performance, illustrating the critical importance of data locality for good performance on current RISC processors. Second, CMMD CVL has the lowest ratio of communication-to-computation performance, which is desirable for communication-intensive programs such as NESL. Third, even when using the coarse-grained message passing of MPI, the modern MIMD platforms generally achieve a lower ratio than the older, fine-grained SIMD CM-2. Note that this would probably still hold true if the hypothetical CM Fortran implementation of CVL existed for the CM-2, since the reduced memory traffic compared to the current Paris implementation would be more than offset by higher floating-point performance. Finally, the communication-to-computation ratio of MPI CVL on the CM-5 is worse than that of CMMD CVL, since their computation performance is identical.

4.2 CVL instruction overhead

Apart from asymptotic performance, we must also consider the fixed overhead of CVL instructions when comparing CVL implementations. This affects the vector length $n_{1/2}$ at which half of the peak performance is achieved. In MPI CVL, there are two significant sources of fixed overhead. The first is the reliance on MPI's scan and reduction functions, and in particular the use of global sums to compare message counts and determine when all messages have arrived. The second source of overhead is the buffering used within MPI CVL permutations to amortize message overhead, which results in peak performance not being reached until all of the buffers sent by each node are full. Thus, given a random permutation where every node sends an equal amount of data to every other node, peak bandwidth isn't achieved until each node has approximately NB data, where N is the number of nodes and B is the size of a buffer.

The overhead of an integer plus-reduce (global sum) function for each platform is given in Table 3. The number of CVL 64-bit floating-point additions this represents is also given, to compensate for different clock rates.

CVL Implementation	Overhead (uS)	FLOPS
SP-1 MPI	750	5900
Paragon MPI	900	2300
CM-5 MPI	2350	2250
CM-5 CMMD	75	70

Table 3: Overhead of CVL sum function and the number of CVL 64-bit floating-point additions this represents (32 nodes).

The factor of 30 difference in overhead between MPI CVL on the CM-5 and CMMD CVL is mostly due to the initial implementation status of the CM-5 `mpich` code. Although `mpich`'s abstract device interface includes support for defining collective functions, these are not currently supplied for the CM-5's control network. Thus, scans and reductions must be performed using point-to-point messages instead, as they are on the other machines where no control networks exist. It is interesting to compare this factor of 30 with the factor of 20 observed when implementing the global product using active messages in CMMD CVL (see Section 3.2).

Since a global sum is always performed at least once by every permutation function, we would expect the combined overheads of the buffering and global sums to result in very high $n_{1/2}$ values for the permutation functions. That is, the asymptotic communication performance shown in Table 2 should only be achievable with a high number of elements per node. Figure 5 shows the communication performance of MPI CVL achieved at various ratios of elements per node, as well as that achieved by CMMD CVL. Again, a random permutation of 64-bit words is used. Note that the horizontal axis has a log scale, in order to more clearly show all the points of interest.

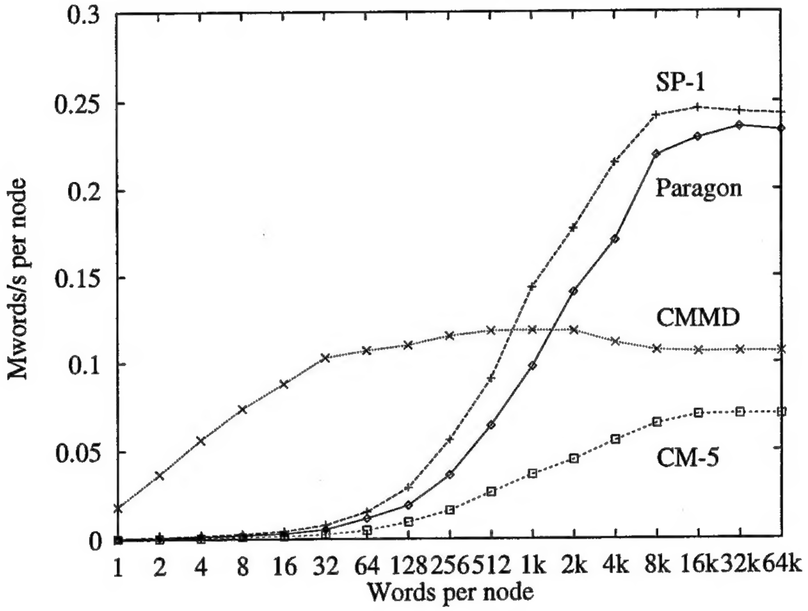

Figure 5: Effect of number of words per node on per-node performance of CVL permutation function (32 nodes, unsegmented random permutation of 64-bit words).

As can be seen, in this example CMMD CVL achieves 50% of its peak performance when there are only 4 or 5 words on each node, and reaches asymptotic performance when there on the order of 32 words per node (that is, one word for every other node, assuming a uniform random permutation). There is also a noticeable cache effect, with performance rising slightly above the asymptote until there are 2k words per node, at which point performance falls again.

By comparison, the $n_{1/2}$ values for MPI CVL on all machines is on the order of a few thousand words per node, clearly showing the effects of higher fixed overhead. Cache effects are not so clear cut.

5 Summary and conclusions

MPI has fulfilled its promise of portability and ease of development, especially in comparison to the communication libraries we have previously used. A full performance evaluation is made more difficult by the fact that the CM-5 is the only machine on which we have both MPI and native CVL implementations available for comparison, and that the native CVL implementation uses a fine-grain message architecture quite unlike the MPI model. However, the current MPI CVL implementation appears to have sufficient asymptotic communication performance to be adopted as the standard release for future coarse-grained message-passing MPPs. The remaining performance problems are in the area of fixed overhead for communication primitives, due to the buffering and reductions explained in Section 3.4. Given the nature of the underlying machines, these appear to be insoluble, and limit MPI CVL's efficiency on small problem sizes.

We separate our comments on MPI into those specific to a particular implementation, and those aimed at the standard as a whole. To achieve high peak performance as well as low $n_{1/2}$ vector lengths for MPI CVL, we want the following from an ideal MPI implementation:

- Full support for the capabilities of the underlying machine, such as the control network of the CM-5.
- Low per-element overhead (which affects asymptotic bandwidth).
- Low fixed overhead (which affects $n_{1/2}$).

The mpich implementation currently appears to come closest to this ideal for most machines, and includes good support for performance visualization and analysis. We are working with the mpich authors to implement the collective communications functions on the CM-5 control network.

In terms of future revisions to the MPI standard, our main request is better support for fine-grained communication, using a deposit model, active messages, or indexed permutations. Such an extension is currently being considered for MPI 2. As explained in Section 3.4, exclusive scans would also be a benefit.

5.1 Future work

As should be obvious from the performance figures in Section 4, MPI CVL's elementwise and communication functions do not approach the peak performance of the current RISC-based MPPs on which it runs. There are a few remaining areas of MPI CVL where incremental improvements in performance can probably be achieved, such as lazy computation of segment descriptor fields, a specialized vector pack function, better tuning of buffer sizes, and using the alternative method of checking whether a permutation has finished (see Section 2).

However, as explained in Section 1, the poor performance of CVL on elementwise functions is due to the current mismatch between processor speeds and main memory bandwidth. Since elementwise CVL functions are essentially single loops over the data, they get no benefit from any cache for large problem sizes, and become limited by main-memory bandwidth. This problem is becoming progressively worse as RISC CPU speeds continue to outrun DRAM bandwidth [1].

One obvious solution is to abandon CVL and the VCODE interpreter, and instead compile NESL down to a low-level language such as C or assembler, which allows us to perform loop fusion and other optimizations [8]. This can be combined with the communication optimizations mentioned above, and with new models for the control and partitioning of nested data parallel programs on MIMD machines. We are now actively working on a system that will achieve this. To achieve portability, and to reduce our dependence on any one machine or manufacturer, it will use MPI as the communications substrate.

Acknowledgements

NESL, VCODE and CVL are creations of the Scandal project, and I'd like to thank all past and present members of the project for their advice and guidance. In particular, Guy Blelloch's NESL compiler and Jay Sipelstein's VCODE interpreter helped uncover many bugs, and Siddhartha Chatterjee and Marco Zagha provided the initial design sketch for CM-2 CVL. I would also like to thank William Gropp and Rusty Lusk for their rapid and helpful responses to `mpich` bug reports and MPI questions.

CVL code and Scandal papers are available on the WWW, at `http://www.cs.cmu.edu:8001/Web/Groups/scandal/home.html`, and via anonymous FTP from `nesl.scandal.cs.cmu.edu`.

References

[1] David H. Bailey. RISC microprocessors and scientific computing. In *Proceedings of Supercomputing '93*, pages 645–654, November 1993.

[2] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, 1990.

[3] Guy E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[4] Guy E. Blelloch and Siddhartha Chatterjee. VCODE: A data-parallel intermediate language. In *Proceedings of Frontiers of Massively Parallel Computation*, pages 471–480, October 1990.

[5] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Margaret Reid-Miller, Jay Sipelstein, and Marco Zagha. CVL: A C vector library. Technical Report CMU-CS-93-114, School of Computer Science, Carnegie Mellon University, February 1993.

[6] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[7] Patrick Bridges, Nathan Doss, William Gropp, Edward Karrels, Ewing Lusk, and Anthony Skjellum. *Users' Guide to* `mpich`*, a Portable Implementation of MPI*, November 1994.

[8] Siddhartha Chatterjee. *Compiling Data-Parallel Programs for Efficient Execution on Shared-Memory Multiprocessors.* PhD thesis, School of Computer Science, Carnegie Mellon University, October 1991.

[9] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In *Proceedings Supercomputing '90*, pages 666–675, November 1990.

[10] Cray Research, Inc. *PVM and HeNCE Programmer's Manual*, May 1993. SR-2501 2.0.

[11] Rickard E. Faith, Doug L. Hoffman, and David G. Stahl. UnCvl: The University of North Carolina C vector library. Version 1.1, May 1993.

[12] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report MCS-P342-1193, Computer Science Department, University of Tennesse, 1994.

[13] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3.0 User's Guide and Reference Manual*, February 1993.

[14] Jonathan C. Hardwick. A proof-of-concept for CVL on top of PVM. Internal report, May 1993.

[15] Peter Mills, Lars Nyland, Jan Prins, John Reif, and Robert Wagner. Prototyping parallel and distributed programs in Proteus. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 10–19, Dallas, Texas, December 1991. IEEE.

[16] Gary W. Sabot. Optimizing CM Fortran compiler for Connection Machine computers. *Journal of Parallel and Distributed Computing*, 23(2):224–238, November 1994.

[17] William Saphir. A comparison of communication libraries: NX, CMMD, MPI, PVM. NASA Ames User Seminar, `http://lovelace.nas.nasa.gov/Parallel/People/wcs/talks/message_passing_comparison.ps`, November 1993.

[18] Thinking Machines Corporation, Cambridge, MA. *Paris Reference Manual*, February 1991. Version 6.0.

[19] Thinking Machines Corporation, Cambridge, MA. *CDPEAC: Using GCC to program in DPEAC*, December 1992.

[20] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, December 1992. Version 2.0.

[21] Thinking Machines Corporation, Cambridge, MA. *CMMD Reference Manual*, May 1993. Version 3.0.

[22] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.